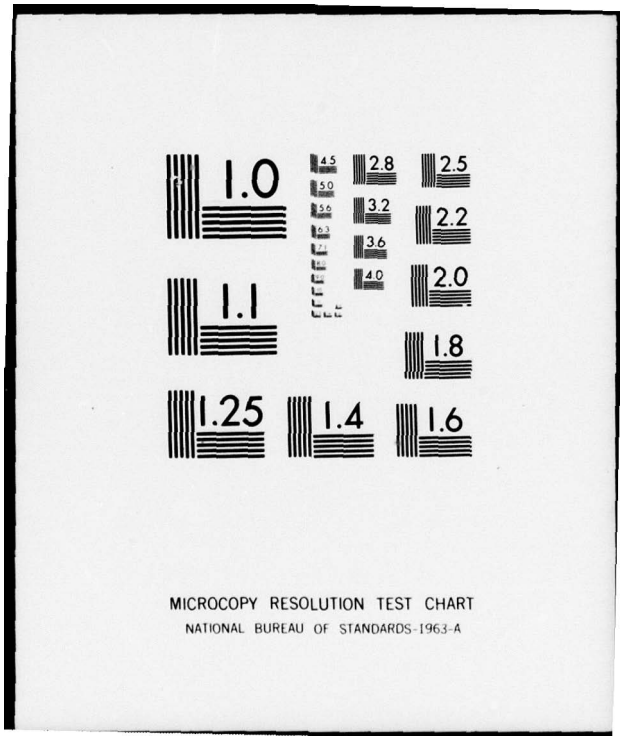
AD-A036 112 ROYAL AIRCRAFT ESTABLISHMENT FARNBOROUGH (ENGLAND) F/G 9/2
AN ESSAY ON COMPUTING.(U)
JUL 76 D M GILBEY

UNCLASSIFIED RAE-TM-MATH-7604 DRIC-BR-54529 NL

| OF |
AD
A036112

END
DATE
FILMED
3-77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

TECH. MEMO
MATH 7604

UNLIMITED

TECH. MEMO
MATH 7604

ROYAL AIRCRAFT ESTABLISHMENT

AN ESSAY ON COMPUTING

by

D. M. GILBEY

July 1976

(14) RAE-TM-Math-7604

ROYAL AIRCRAFT ESTABLISHMENT

(9) Technical Memorandum Math 7604

Received for printing 30 July 1976

(18) DRIC

(6) AN ESSAY ON COMPUTING.

(11) Jul 76

by

(10) D. M. Gilbey

(19) BR-54529

(12) 25p.

SUMMARY

An essay is presented on the nature and difficulties of working on or with computing systems. It is hoped that the intelligent layman can read past any unfamiliar jargon and still be convinced, along with the 'occasional programmer', that modern computing is mainly about the creation and management of the complex.

310 450

LB

CONTENTS

1 INTRODUCTION

Vast changes have been made in digital computing practice over the last few years, and the pace of change is still rapid. The aim of this essay is to present a somewhat academic view of the processes involved in the exploitation of general purpose digital computers and of the attendant difficulties. It is assumed that the reader has had some contact with digital computing but is left unsatisfied by more popular accounts either because these stop short at binary arithmetic and elementary programming, or because they are accounts of the wonders of particular systems with no recognisable generality or sense of direction.

The rate of change is in itself a cause of difficulty, requiring rapid re-education to changing roles even at high management levels. Traditional mechanisms of adjustment are overstrained, and knowledge and skills can rapidly become irrelevant to real needs. The difficulty is compounded where it is not recognised as such. A rapid identification and recognition of new problem areas is required.

However, wider issues such as computer-induced unemployment, privacy of computerised information, status and career structures for computer staff etc. are not the concern of this paper. Such problems occur in any radical change of practice, and whilst a sound appreciation of technical possibility, likelihood and reliability is needed, the real requirement is political, legal and managerial machinery capable of coping reliably and expeditiously with the pressures generated. The concern here will be with recognition of problems faced by those who actually design and use computer systems.

Computing as an activity has progressed rapidly from a demonstration in a research laboratory to a major industry whose products affect our daily lives in a great and increasing variety of ways. This phenomenon needs to be appreciated in some way or another by all concerned. It cannot be dismissed as change for the sake of the technologist at taxpayers' expense, nor can it be understood in terms of 'superfast idiots' or bills for £0.00, nor yet by an appreciation of binary arithmetic or of the physics of computing machinery. It is suggested here that the truly inescapable and perhaps characteristic feature of modern computing is complexity. All intelligent people are called on to think about, and to manage, complex situations but in computing activities this is the central and crucial problem.

Change often exhibits the limited foresight of those responsible. Planning action may precede decision but experts often disagree. In a youthful science the experts may feel their responsibility for the rapid development of their profession and its possibilities more keenly than their responsibility to the wider community for a wise decision on a specific issue. This danger will not abate until people acquire a good idea of the kind of thing that is liable to go wrong, and until codes of practice evolve and the profession generates a body of reliable knowledge and of attested and relevant expertise.

This all takes time, so it is not surprising that at first the computer man may be dangerously deluded about the wider business his system is supposed to serve, while the man responsible for implementation of the system is unaware of the planning checks he should be calling for and at the mercy of the designer on matters of reliability and cost estimation. The researcher who built, single handed and quickly, a pilot model to demonstrate the principle a 'long' time ago often fails to appreciate the rise in complexity and liaison activity required in going from demonstration model to operationally reliable version and he probably feels that the system designer trying to follow his footsteps is professionally incompetent. There are also more technical reasons for increasing complexity and some of these will be explored later. First it is necessary to expose some of the more important and general features of digital computing machinery, and of its exploitation, and the nature of some of the problems raised.

2 USING A COMPUTING SYSTEM

Assuming that there is a good case emerging for doing a given job at all, it may be done by computer because it is not feasible to do it in any other way or because doing it by hand involves tedious book-keeping or repetitious calculation or because a computer process is perhaps cheaper or more reliable or more responsive and adaptable than alternative means. Any such claims need to be re-examined periodically as the task and/or its associated programs become better defined, developed and tested. However the need for this continual review of hopes, assumptions, deductions etc. is all-pervasive in what follows and need not be over-stressed here.

A computing system is usually a completely general purpose facility, capable of performing, within its considerable resources, any prescribable operation on some body of data to be supplied and possibly some data already stored, perhaps to modify the stored data and to present any desired result as output. The user on the other hand has a specific task in mind and therefore he is

required to specify just what it is that he wants the system to do. At some stage a computer program has to be prepared which, together with some instructions for the preparation or acquisition of input data and for the control of the job in the system and for the handling of output, enable the desired result to be obtained.

Two immediate topics are raised by this - first the nature of the language most suitable in which to convey the various instructions and secondly the processes by which the user with a need to be met reaches the stage of having a suitable program or 'job' available. We will here concentrate on the second question.

It is possible to identify a number of stages, between having a need and having a computer job designed to contribute to the satisfaction of that need. Not all stages are important in all jobs and even fewer are explicitly separated in the activities of many programmers, but all are potentially important, particularly to the general scientist or engineer served by a centralised computing service.

(i) Selective attention: An employee is usually paid to set aside as far as possible his more personal and extramural interests during work time and to devote himself to some area of need in the organisation he serves. This often allows still a considerable further choice of strategy and approach and selection of areas of prime interest.

(ii) Problem identification: The next stage in the process of successive abstraction from 'real life' complexity, i.e. of mapping from a richer thought-space to a more manageable one, is to define a problem. At first one merely experiences a recurrent difficulty - recognition of the 'fact' that history is in some sense painfully repeating itself is a vital step which already does some necessary violence to the 'whole truth'. Continued experience or thought will indicate that a few factors or features are more or less invariant as other things change (in passing from one manifestation of the problem to another) and these can be used predictively. Others are recognised as major determinants of the compromises to be made or the measures and designs to be adopted. Attention is concentrated on these important features and a problem is obtained which can be described in terms of a limited number of factors and constraints, the manner in which these factors interact, and the way in which they influence the outcome. One could be said to have solved the problem when one has available an easy means to produce a generally satisfying outcome from a knowledge of these factors and relationships.

(iii) Idealisation: In most scientific work a means will be sought to represent the factors and relationships within some coherent branch of mathematics, i.e. as numbers, functions, sets, networks, grammars, etc. Even in administrative work it is often necessary to define carefully what is to be taken to be a person's age or pay so that it can be represented unambiguously as a number which can enter some formula or decision logic. Hopefully the result is a model situation in mathematical terms which more or less adequately formalises the essential problem. It should perhaps be emphasised that such entities as names, book references, propositions, etc. etc. are perfectly acceptable to mathematics provided that rules are established which define the result of any required comparisons between or operations upon the entities.

(iv) Symbolic representation: This stage is not an abstraction from the mathematical model but a simple one-to-one mapping of it into a concise and comprehensible notation to facilitate grasp of the mathematical nature or 'form' of the problem. Single characters from various alphabets are often used to denote factors, system state vectors, operators, etc. etc.

(v) Analysis: If the problem is sufficiently simple, it may be amenable to analytical solution, with the hope that production of specific solutions may be reduced to simple evaluation of mathematical expressions involving the available input factors. There is usually also an enormous benefit because the functional dependence of the solution upon the input factors can be appreciated mentally in a very general way. More often, however, the most that comes from the attempt is a suggested procedural or algorithmic approach to the desired solution, with an attendant need for further analysis of parts of the algorithm perhaps using specific input values because one cannot find a reliable general analysis and one is forced into exploratory survey. Even if the problem is, for example, stock control there is analysis required. This may be overtly and numerically mathematical, e.g. probability of 'stock-out' with given control parameters or likely performance of a sorting algorithm with the kind of data anticipated. It is more likely, however, to be the programmer convincing himself that his algorithm is proof against various 'odd' exceptions to his general logical analysis, which exceptions might, for example, arise from Easter or a strike or an error recovery procedure, or some combination of such things.

(vi) Symbolic process: The analysis phase may give rise to a definite symbolic process, perhaps represented as a high-level flow chart, decision tables or master routine coding, which it is desired to use with various sets of input data.

(vii) Source program: The symbolic process must be embodied in a program which complies with the rules of some computer language. This may require considerable elaboration of the symbolic process, for instance to explain to the computer how to invert a partitioned matrix or to sort variable-length records until their key fields are in lexicographic order. One will have to specify in detail the layout required of the printed results. Also a mapping is made from the symbols, factors etc. of the problem on to a set of 'identifiers' with which are associated 'types' such as 'integer' or 'real number'. The language used may perhaps not recognise a type such as 'string' (a row of characters of arbitrary length) so that the programmer is forced to represent and handle such an entity in some artificial way. Equally, he may have to choose an identifier 'alpha' or even A_1 to stand for α in the symbolic representation. This shift of representation to an artificial, possibly long-winded and certainly less-comprehensible one gave rise to the term 'coding' as an alternative to 'computer programming'. Modern high-level computer languages have done much to minimise this undesirable feature, but 'code' remains the American term for a computer program.

(viii) Job control instructions: The computing system or its operators will need to be told what language has been used, how to identify the various input or output 'documents' (card decks, paper tape rolls, magnetic tapes, print, etc.) and which source program 'channel' is for which document. Alternatively these matters may be Installation Standards or given by the system as default options. There may also be instructions concerning alternative actions to be performed if certain events occur in the course of running the job. Modern computing systems are run by an 'operating system' program which takes over from the operators the task of accepting and acting upon most of these job control instructions. This means that a 'job description' written in a 'job control language' (JCL) has to be provided in computer-readable form. Alternatively the user can type the necessary instructions or commands at an on-line teletype or display keyboard etc. as the job proceeds. It is not unknown for the programmer to experience more trouble in writing in the JCL than in preparing the source program.

(ix) Consideration of results: The computing system will produce some result from the job. This may be trivial, diagnostic of command errors in the JCL, or compiler diagnostics indicating transgression of the rules of the source language. Alternatively the program may compile and run but commit a run-time or 'execution' error such as trying to divide by zero. Each of these possibilities will frequently return even a good programmer to an earlier phase of the work,

and the earlier the phase the more work is occasioned in rectification of the error. Thus even when the desired process with correct data has been successfully modelled it will often transpire that an earlier stage of the work requires revision before anything of wider import can usefully be gained from the study or before the process can be regularly used to good effect. A computer user typically has a wide range of choice over how much effort he devotes at each level of the work and when, and his usefulness is strongly conditioned by the aptness of these choices.

Computers are often used or worked upon by teams. One might find a manager dealing with stages (i) and (ii), an analyst dealing with (iii) to (iv) and a programmer or coder coping with the rest. Or, indeed, there may be several or many people working together at various levels and on various facets of the work. The effort needed to produce some program systems is measured in thousands of man-years.

We have just structured the activities of a computer user into a number of levels. The next section considers design levels within the computing system itself. Part of the object in doing this is to expose some more of the rich tapestry of computing and part is to remind the reader also that one of the simplest tools in dealing with a complex situation is to try to stratify it into recognisable layers. In passing, let us note that one of the results of this very necessary strategy is that it is possible to be a computer professional of high calibre nowadays without understanding hysteresis or knowing what an electron does.

3 COMPUTING SYSTEM LEVELS

A report on the work of the National Research and Development Corporation defined the following levels of complexity, at any one of which innovations could be made: service/system/sub-system/component/material. A librarian might comment that there are several classification schemes other than by level of aggregation, and indeed we may be more concerned to identify functional responsibilities at the higher levels. At the lower levels there is no need to say much here or to worry over such distinctions and for an initial look we may propose the levels:

PEOPLE (ORGANISATION OF USERS)
(ORGANISATION OF SERVICE STAFF)

ARCHITECTURE (SOFTWARE)

ARCHITECTURE (HARDWARE)

TECHNOLOGY

PHYSICS

MATERIALS

This classification has the merit, at lower levels, of dividing responsibilities between recognisable disciplines. Thus the physicist is expected to identify the physical phenomena to be exploited in devices, and the phenomena which limit performance. He gives guidance to the (computer hardware) technologist on what is possible and to the chemist on the chiefly desirable properties of materials to be prepared. Devices are fabricated and brought into mass production which offer ever increasing speed and reliability to the digital logic designer, who can then do his work in terms of gates, bits, highways, registers, storage media, etc. - hopefully with minimal need to understand the physics and chemistry on which his work depends. The subject of hardware architecture will, unfortunately for our attempted stratification, involve physics in matters such as design of back wiring, thermal design, 'flying' magnetic read/write heads close to moving surfaces etc., but the hope is to present to the software designer, by means of microprogram units, address translators, store access mechanisms, arithmetic units and the like, a simple specification of a set of machine instructions with defined interpretation in terms of the change of content of a few store locations. Some bits of an instruction word will select which operation is to be performed and others will help to select which register(s) and/or which word of immediate access store is/are to be involved.

Popular accounts of computers are often pitched at around this level, explaining how arithmetic can be mapped on to logical operations with binary representations of integers. They will often go on to explain how simple instructions can be put in sequence to achieve more complex results, for example, the computation of the square root of an arbitrary number. However, it is above this level that complexity really explodes. The user of a computing system, which may have billions of words of data in its store and millions of words of instructions, needs an interface which will insulate him from all but that part of all this complexity that really concerns him. Furthermore the users may be many, with diverse and conflicting interests and requirements, and the management and operation of the facility requires another interface which must be

defined at a suitable level for human beings to operate. Between these interfaces and the one represented by the specifications of the set of machine instructions lies the realm of software architecture, and here the lines of functional responsibility within the system begin to diverge even more radically from the classification by aggregation level proposed above.

The user will normally express his requirements, as we have seen, in a 'high level language' (HLL) such as FORTRAN, COBOL, ALGOL ....., supplemented by a JCL such as the GEORGE 3 command language, system control language, OS360 JCL ..... . It is the task of the compiling system to convert the former into an equivalent program of machine instructions, whilst the operating system software deals with the JCL. There is little real agreement over a proper division in detail of responsibilities between compiling systems and operating systems. A consequence is that it is common to find certain functions duplicated or even triplicated in the actual operation of a user's job. Indeed there are those who doubt that these sub-systems should be distinct, and in particular feel that the HLL and JCL should be merged into a single language.

It is not proposed to say much here about the organisation of people around the computing system as this is dictated largely by the wider enterprise which the system serves, and by the requirements for computer service which its work generates. However, it will be proposed that in their general nature the most pressing difficulties facing the designer and user of computing systems are just the same as those facing a general manager or those which arise in the design and assessment of any complex system or those which arise in R & D. The task is to generate a partially controlled complexity, through progressive understanding and organisation, out of potential chaos. The task is that of any thinking person, and is never completed, because circumstances and possibilities are always subject to uncontrolled changes and the time allowed is never enough to produce a unique demonstrably appropriate response at all levels. The next section explores in more detail the nature of chaos and complexity, and the weapons of organisation which are used in the computing context and in other contexts mentioned above.

Before leaving the matter of levels, it is worth remarking that present day operating systems are so complex that a number of levels can be discerned within the separate functions of the software alone. An example is the scheduling function (see Fig.1). Typically, the various resources of the system such as the machine-code processor (CPU), peripheral control devices, store space, etc., will be subject to a predefined or single-program sequence control only for a very limited 'time span of discretion' before a lowest level of

software controller is called in to time-share the resource amongst the competing processes which want to use the resource. This controller's simple priority-ordered list of processes to be served may be governed by a higher order control task of greater comprehension which interferes correspondingly less frequently to try to meet objectives in terms of the less-discriminating 'overall computing power' to be assigned to a wider list of jobs given to it by the highest order controller. The highest order controller may perhaps be invoked only once every few seconds (or every million or so machine-level instructions) and have overall charge of the acceptance by the system of jobs and of scheduling instructions from human operators and users. These instructions would pass through, and be routed to the scheduler by, the JCL interpreter. Responses meant for human consumption may well be generated for the scheduler by a central message organiser and so on.

In complex control systems, as in human organisations, there can be ambiguities between 'brotherly co-operation' and hierarchical control. It must (often) also be possible for a low level controller to refuse or delay a request from above, perhaps because a device has become inoperable. This makes for a considerable variety of possible response to a command, and an equal or 'requisite variety' of behaviour on the part of the higher level controller.

The number of possible states and behaviours of the overall system is subject to the usual combinatorial explosion and so design, even if clear in principle, needs the backing of simulation and/or empirical study. Similar observations can be made about large applications programs or suites of programs as well as of operating systems. In complex systems it is important to be able to describe and investigate the behaviour at all appropriate levels - a study of pulse shapes will give little or no information about malfunction in a software control module. Similarly a study of overall workload behaviour is a poor (but sometimes the only initially available) diagnostic of a time sharing disorder.

Each of the boxes in Fig.1 represents a body of program code which processes data, some of which it shares with connected operating system 'modules'. Whilst scheduling is perhaps the most clearly hierarchical function in the operating system, its complexity is quite typical of the code to be found in large integrated systems. Perhaps it should be emphasised that scheduling is only one of the many functions of an operating system. Large parts of the operating system may well be treated at the lower levels of control simply as other programs competing for the use of the processor. Thus we have a program controlling its own access to the processor, to store space, etc. The system is (in part) its own metasystem, to use Stafford Beer's terminology.

In thinking of system software, it is therefore often necessary carefully to distinguish between the 'objective' and 'subjective' roles of the same piece of code. When the code is being obeyed, and not being accessed merely as an object or body of data by another module, it is a special purpose machine and moreover it is the agent, within the computing system, of its designer. It becomes quite false and artificial to avoid anthropomorphisms such as "... at this stage the compiler knows ... and is trying to ...". Of course, the perceptual and cognitive mechanisms provided by biological evolution can far outstrip the performance of any existing computer code at making sense out of chaotic incoming data of enormous potential variety. Nevertheless this is the essential nature of the source language analysis function of a compiler: the stream of characters it examines could turn out to be gibberish, but is more likely to be an approximation to one of an infinite variety of valid programs.

Fig.2 indicates that despite great efforts to present the user with a simple interface, his interactions with the service are often conducted at a variety of levels of detail. It could be argued that there is no justification for putting applications programs lower in level than compiling systems, since one man's compiler or operating system is another man's program. But no system of classification ever is satisfactory and Fig.2 will do for simple, investigatory, ephemeral application programs such as are common in scientific and engineering use.

4 CHAOS AND COMPLEXITY

So far, we have taken a glimpse at the activities of a computer user and a glimpse into the world of hardware and software design. The message to be conveyed is that third generation computing systems are complex inside, though they try as far as possible to present a simple and manageable interface to the user. Furthermore, the difficulties faced by programmers and designers and users are less to do with learning a high level language or binary arithmetic than to do with the management of complexity or just plain thinking.

It is a theme of this paper that the task of writing a computer program is tackled in the same way that one would design an organisation or a weapon system or prepare a case or write a report or construct a mathematical proof, etc. It is almost wholly an intellectual exercise in which a structure must be conceived which can effectively link the need to the resources available. A schematic view of the problem is given in Fig.3.

Generally there are two approaches - the 'top down' and the 'bottom up'. The top down approach starts with the need and tries to find a number of simpler tasks which if accomplished together will ensure satisfaction of the need. Each of the simpler tasks is further refined in a similar way until one arrives at a multitude of tasks each of which can be accomplished easily. In the bottom up procedure one experiments to see what can be done easily and what is the effect of combining the easily done things in various ways. The result is a rich armoury of fairly simple components, results or subroutines which can be combined into more complex results or subassemblies having properties that should be useful in tackling the overall task. The building proceeds until hopefully one finds a structure designed to meet the need.

There are those who preach the top down procedure as the only proper one worthy of the name 'design' and those whose practice appears to be mainly bottom up or even serendipitous. The conflict also appears as 'need-pull versus technology-push' in talk of R & D matters. Those engaged in actually designing things will recognise that bridges are often best built from both ends hoping to meet in the middle, as in the numerical solution of 'jury' problems. Also the first bridge that actually works tends to be ramshackle through poor foresight or navigation during its building, and it often pays to rebuild it more elegantly and strongly if it is to carry much traffic.

The initial leap from ends to means must be intuitive or inductive. It may be as inconsequential as the ramblings of a drunken poet, though experience will help to discourage the choice of some of the seductive blind alleys. The leap leaves a tangle of gossamer - frail links which must be refined and strengthened until even a sceptic can be made to admit that a credible connection exists. Corners must be cut and irrelevant links discarded. All of this is the automatic or unconscious result of a lifetime of purposive thinking, but there is never any assurance of success. The whole enterprise can be impossible.

In programming, the real feature of the 'traffic' which the program has to sustain is the essential variety in the data which may be encountered in use. In some sense this defines an 'operational envelope' for the program, and a complex program will often succeed for most of the data cases which are presented to it but fail on a few special ones - perhaps because an iteration diverges. The failure may be catastrophic if the programmer was unaware of the possibility, or did nothing about it. The more obvious element of 'traffic' is the frequency of use of the program and the sheer amount of the data it is called on to digest. If this is great it may be worthwhile to redesign the program, although it is

already effective and reliable, to make it more efficient. Unfortunately efficiency or minimum consumption of resources often conflicts as an objective with simplicity and reliability and sometimes its pursuit in the name of craftsmanship wastes more in terms of programmer's time and in development and re-run time than is saved in terms of machine resources. It must be allowed, however, that a bad programmer can write astoundingly inefficient programs which are also far more complex than they need to be.

The programmer's task, then, is to discern or to create and impose a structure on the job to be done, on the code to be written, and on the internal and external data with which the code deals. The distinction between 'discern' and 'create' is probably unreal - even perception is creative, the structures we build can be well or ill fitted for their purpose. In programming we would ideally like to arrive at a series of code 'modules' interrelated hierarchically at a number of levels, each having a simple specification guaranteeing that it will not tread on its neighbour's toes and that the overall job will be done. Then one can concentrate on producing code to satisfy the simpler module specifications. Even if all of this is not done explicitly, at least the programmer will have some glimpse or intuition of such a structure in his mind when he begins coding.

Despite the aspirations of the top-down school, it will generally be found that the structure initially conceived will change as one tries to encode the modules. A module will prove impossible to make, or it becomes evident when dealing with the detail that there are system problems which are not catered for in the specifications. A different structure begins to look more attractive and perhaps more efficient too, and we may even begin to look at the overall problem in a new light. Nevertheless, however limited one's foresight in planning may be, some planning is desirable. It is advisable, for instance, to consider which data components need to be used by which code components and what lockout or sequencing restrictions may need to be imposed on such accesses.

This then is the weapon of organisation by which chaos is reduced to a partially-ordered complexity. The weapon of standardisation will be mentioned later. One might draw an organisation chart or tree showing which subroutines are parts of which segments in which programs and which programs make up the entire suite. The tree is one commonly used structure. As with human organisations, at whatever level is of interest it is often necessary to draw lines of one kind and another between the modules to indicate the essential nature of the

transactions required in addition to the hierarchical ones. This structure would form a network or graph. Mathematics provides many other useful structures such as vector and function. Commercial work contributes the file/record/field structure. Library practice suggests structures for marking, parking, and picking items in large collections. All of these, and many more, structures are commonly given to computer data through the use of suitable conventions.

Shannon's theory of the selective power of messages indicates how the potential information-bearing capacity of a stream of characters may be diminished by noise and redundancy. The structure of the data, which is known beforehand to sender and receiver, or may be sent with the data, is redundancy in so far as it limits the potential variety of what can be transmitted. In computing it is used at low level to protect against corruption by noise and at higher levels to facilitate recognition and analysis of the message.

Both sender and receiver may be in the computing system, for example when one procedure gives data to another or when a data base is updated by a program. However, we are here more interested in a man's problems when he tries to make sense of programs and data. A programmer is continually doing this in the course of his work and anything that can be done to make programs and data more lucid is usually well worth the effort.

In reading any kind of analysis or case one is assailed by doubts that disrupt the desired smooth progress to conviction - does this function necessarily possess a Fourier transform - will McDuff come before Macdonald, what about Ångstrom, Tschebyscheff and Chebyshev? Progress is rapid when these doubts can be put on one side or disposed of rapidly - "we agreed to use the telephone directory order" - it's all right, that is discussed in Appendix B" - but as soon as a sizeable diversion is created a lot of mental work is occasioned to re-establish the context of thought where one left off. This disruption by interpolated activity is well-known to programmers, managers and mathematicians alike.

In programming one may be coding a particular module and part of the context of thought is in a sense the specification of that module. This specification may or may not exist on paper, it may be clear or cloudy in the programmer's mind. At all events, it is most unlikely that it will be so manifest that the programmer is not occasionally diverted into clarifying the detailed relationships between this module and others, or linking this module's detail with the

higher level abstractions that the program is meant to embody. If he is not a very experienced programmer he will also from time to time have to pause to consider the precise effect of such and such a command or statement. All of these uncertainties absorb even more time and effort if they persist, as they usually do, into the program testing phase.

Clearly, these problems are unavoidable. Equally clearly they can to some extent be minimised by explicit, clear and easily referenced documentation. Nevertheless much of the context material will not be explicitly defined or referenced in any real-life project, and on some facets the source program itself would be the easiest document to use. In considering the nature of programming it is crucially important to remember the implicit, unstated structures which help to relate the program to its problem context. These may very properly be unstated because the programmer knows what he is doing and does not expect anyone else to have to understand it, or because they form part of the general usage, custom and technique of people doing that kind of work so that it would be irritating, gratuitous and pedantic to expect anyone to read it.

Matters of custom and established technique can and do get enshrined in explicit standards, or more accurately a proliferation of standards appropriate to the different circumstances of diverse groups. On the other hand, the programmer who uses the identifier 'M' in his source program to stand for Mach number and never actually records this correspondence may be on more doubtful ground. If he is a scientist or engineer engaged in an ephemeral calculation he may never have to read his source program again (when the correspondence has been forgotten). Also, it may be that a restricted context has been established for the program such that only a few factors are involved, so that the one likely to correspond to 'M' is easy to spot. On such grounds he may justifiably use a cryptic identifier in place of the more explicit ones available in modern high level languages, such as 'MACH' or even 'MACHNUMBER'. He will benefit from brevity just as the mathematician benefits from working with single-character symbols. On the other hand, it is surprising how valuable it often is to be able later to read an old source program quickly and easily. This demands the use of more explicit or mnemonic identifiers, or 'comment' annotations or supporting documents describing the program.

Groups of people differ in their familiarity with aspects of any given context, so it is unsurprising that a report which is a model of clarity and exposition to one reader is tediously verbose to another and yet impenetrably

or aggressively terse or jargon-ridden to a third. The same applies to style in mathematics and programming. The essential task is to communicate something which is known only to the author or source, within a context of what is understood by and agreed upon by both the originator and the intended recipient(s).

The message is not that men and machines are just the same, only that the morphology of some of their respective problems of communication is recognisably similar. Of course, human communication is complex and its explicit objective content may have little relevance to the needs of either party. The important message is often latent in the style or the very fact of the communication.

5 STANDARDS

To adopt a standard of any kind is essentially to limit the designer's freedom of response to his problem and to make it, in all probability, impossible to produce a solution as well adapted to the need as could be produced without the constraint.

Nevertheless in all design fields, including programming, designers do use standard components, carefully disciplined terminology, and standard forms of description and specification. The advantage in general is that a cheaper and more reliable answer can be got in limited time. The designer does not necessarily have to understand exactly why the standard component is made the way it is, only how it behaves as an entity. He is more likely to understand it as a complete entity because he will have used it, or have seen it used, in other contexts.

A further and very important advantage comes with the standardisation of interfaces rather than components: a range of different components built to the standard are interchangeable. Thus one may standardise at machine-code level for a range of machines of widely differing technologies; then programs and machine-code programmers can be moved from one machine to another, or an old machine can be replaced by a faster one without requiring that all the programs be rewritten and the programmers retrained.

This aspect of the 'portability' of programs and programmers and the interchangeability of computer systems is so important that there is much international activity aimed at defining and maintaining standards for high level languages, etc. A standard at this level can of course hope for a much wider coverage with correspondingly greater benefits. As well as such formal standards there tend to be *de facto* standard practices at all levels on what is agreed among some

group of practitioners to be good practice or what has been laid down by relevant authority as a standard in some particular project or activity.

However, although most standards in computing are designed to facilitate transfer or exchange of equipment, data, programs or people, the theme of this essay is complexity. The adoption of standard notations, language, method, subroutines, etc. enables the programmer to shrug off (and to leave undocumented) uninteresting decisions which, it is to be hoped, have only a very minor influence on the optimality of his solution. He is then free to use his mental capacities to deal with more important issues more thoroughly.

The success or otherwise of this manoeuvre depends heavily on the amount of detail which needs to be mastered in the specification of the standard. Often the programmer will complain, and sometimes rightly, that it is more trouble to read the specification or manual describing the standard article, than to make his own version. Similarly many a scientist will undertake a small investigation himself rather than spend days in a library looking for relevant literature. In both cases there is the danger that what is eventually found will turn out not to be quite suitable after the effort has been expended, and there is the loss of that personal understanding and assurance that comes of doing it oneself. On the other hand, progress will always depend in part upon finding people and sources that can be relied upon.

In turn, the complexity of the specification depends on the generality of the proferred article in two ways. It can be too complex either because the article is too generally-applicable for the context or because its generality is insufficiently wide. Thus it would be unfortunate to have to use a subroutine capable of computing the hypergeometric function ${}_pF_q$ in order to get arc sin (Z), i.e. $Z\,{}_2F_1\left(\frac{1}{2}, \frac{1}{2}; \frac{3}{2}; Z^2\right)$. It would be unfortunate in the other sense if the subroutine was subject to failure or catastrophic loss of accuracy in several small regions of the complex argument plane. In the first case one has to specify too much uninteresting data and in the second the analysis is unduly complicated and/or one must provide 'alarms' in the program to guard the forbidden domains, and specify what should be done if the program attempts to compute with a forbidden value.

6 GENERALITY

Of course, the search for generality is basic to mathematics, to science and to thought itself. We seek the uncluttered and simple statements that

remain invariantly true over as wide as possible a universe of discourse and which can yet correlate and account for as much as possible of the variety of our experience.

In this way we simplify experience and link it to the body of public knowledge. If the structure so built is good our powers of perception and thought are enhanced and our powers of communication also, provided the other person (or machine) is possessed of similar structures and relevant public knowledge. If the structure has little enduring utility, it will distort perception and reactions will be inappropriate.

The concept of orthogonality is vital in the reduction of complexity. If a machine instruction can be separated into parts which independently specify the operation and the operand it means that the space of possible instructions is a Cartesian product of two much simpler spaces. The 'dimensions' of operation and operand are 'orthogonal'. It need not have been so: a perverse designer could have specified that the first machine instruction meant operation 23 with the number 12, the second meant operation 2 with the contents of store 6 and so on.

In work on programming languages there is a similar drive to combine a few simple primitives with a few straightforward and logical rules of elaboration into a language of the expressive power necessary to be able to define any desired computation economically and with clarity. The aim might be said to be that of changing programming language from a collection of unrelated tools into a coherent system which can be studied as a branch of applied mathematics, enabling more rigorous comprehensive and general deduction of the validity and effect of the statements and programs which may be uttered.

7 AMBITION

Ambition is the downfall of many a programmer. As we have seen, programming is largely about the management of complexity, and the amount of complexity to be coped with is often very much under the individual influence of the programmer. It can escalate wildly as one seeks to include more factors in the problem, or as one tries to make the program more general or more 'intelligent'. It is thus very easy to bite off more than one can satisfactorily chew.

The general tendency is to choose a problem of one's own size to tackle, and the allowance made for errors of judgment is not always sufficient. There is, however, an optimum to be sought, since if one only tries the trivial then

much of the potential benefit from using a computer will be lost. Whether the programmer or designer tries too hard or attempts too little, computation tends to get a bad name. The call is then for tighter management control.

This evolution in affairs sometimes gives rise to the peculiarly unsatisfactory position that one group of computer people is disciplined but unimaginative while another is creative but unreliable and the two groups never really communicate. The difficulty is not special to computing. It is the same problem which tended to divide research from industrial production to the detriment of both. Once again compromise is inevitable and a willingness to learn of the sensitivities and difficulties of other people in other contexts is far more valuable than a partisan crusade.

8 CONCLUSION

It is hoped that the foregoing will have convinced the intelligent layman or the occasional programmer that the problems of working with computer systems are the same in essence as those facing other people. The thesis is that the real nature of computing work is the creation and management of complex systems in circumstances of rapidly changing needs and possibilities. Computing is no longer a branch of mathematics, physics, management or engineering. It can respond rapidly as an art or as an industry to changing needs, but key questions are often of the form "if it takes one man one day to solve a problem, how long will it take a hundred-man team to solve a problem which is ten thousand times as complex?".

Acknowledgments

Messrs. J.H.B. Smith, D.E. Williams and L.J. Richards have made many suggestions for improvement of the text of this essay.

Fig.1

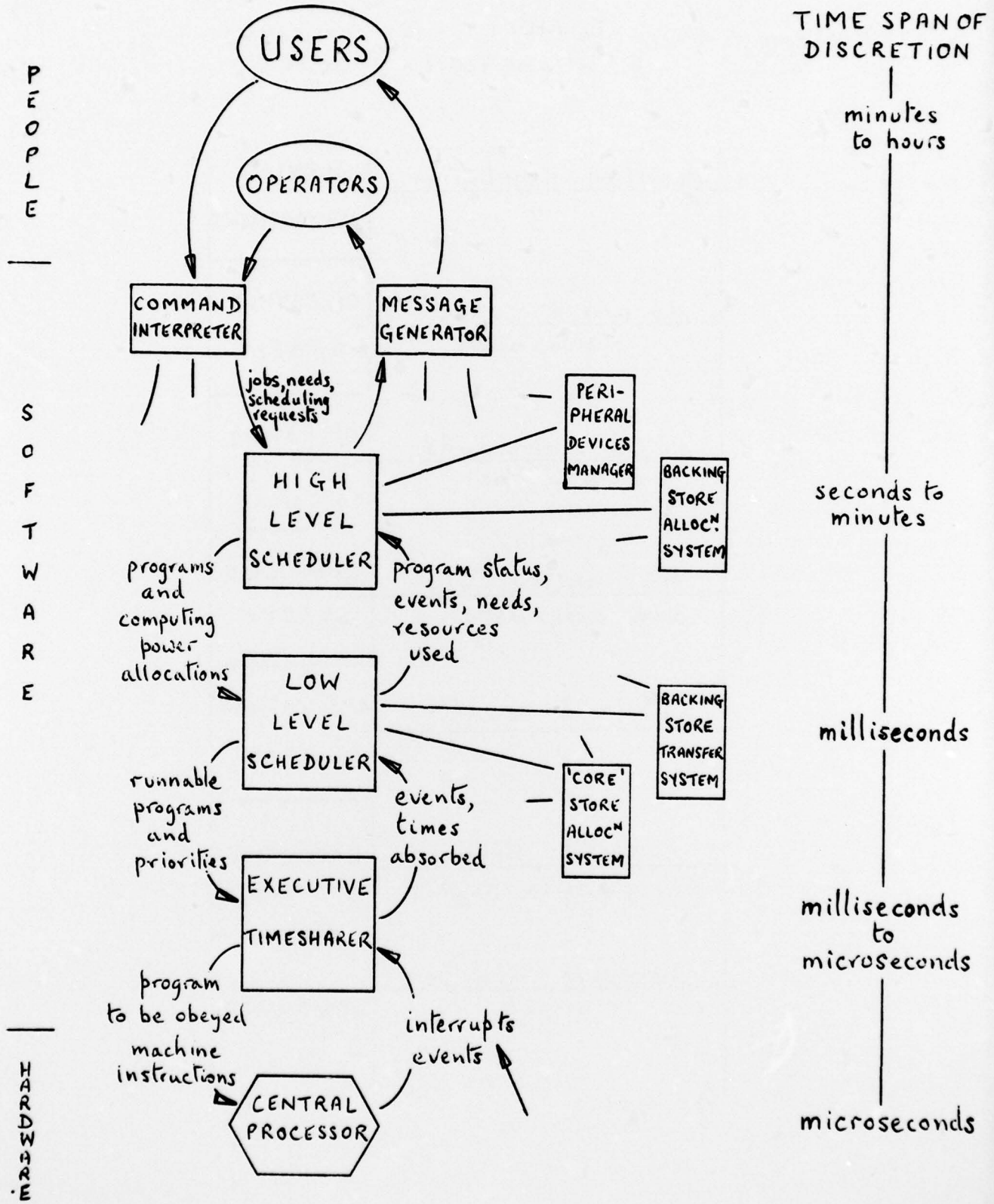


Fig.1 Levels of control over processor use

Fig.2

Math
7604

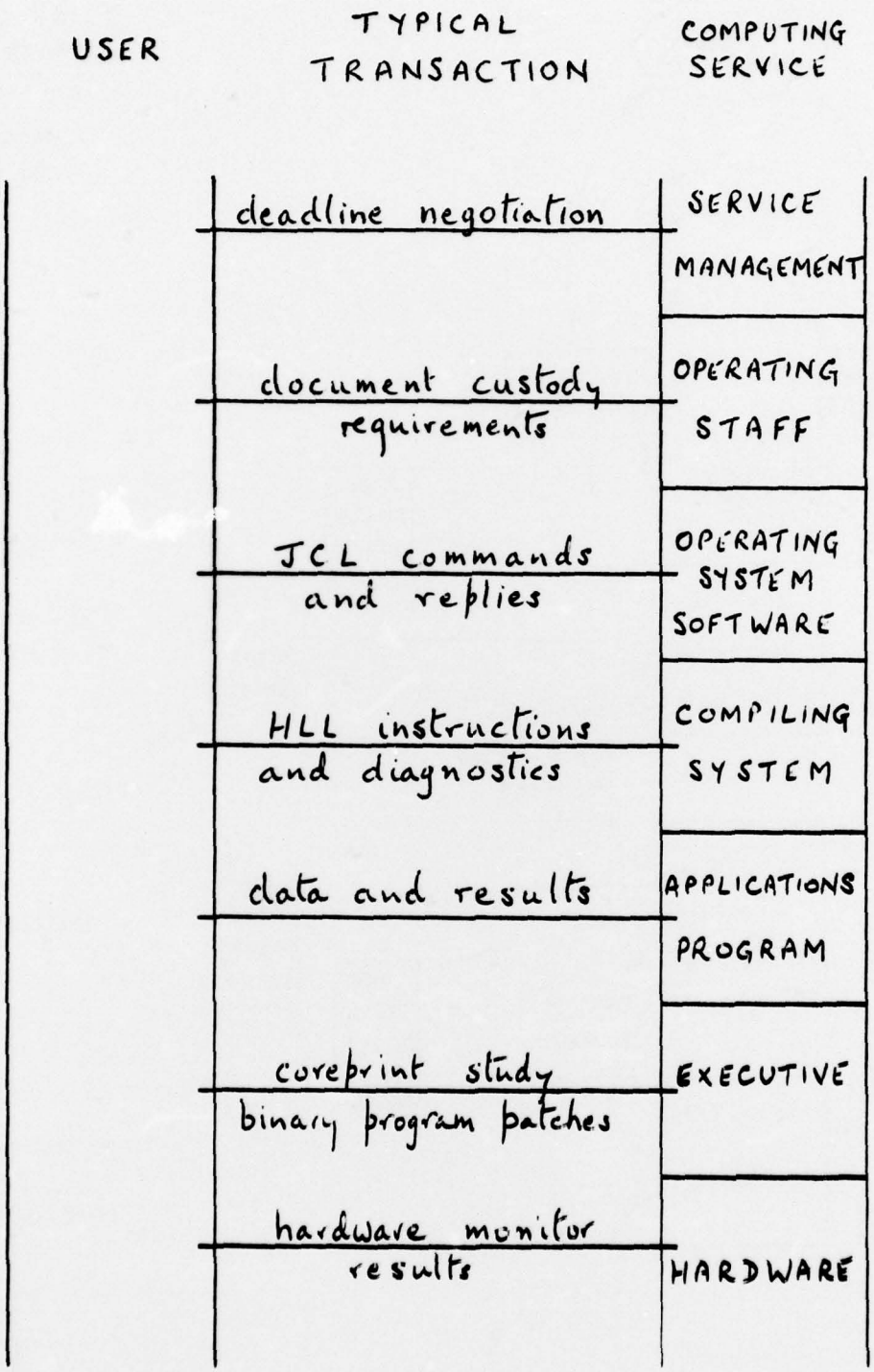


Fig.2 Possible user/service interaction levels

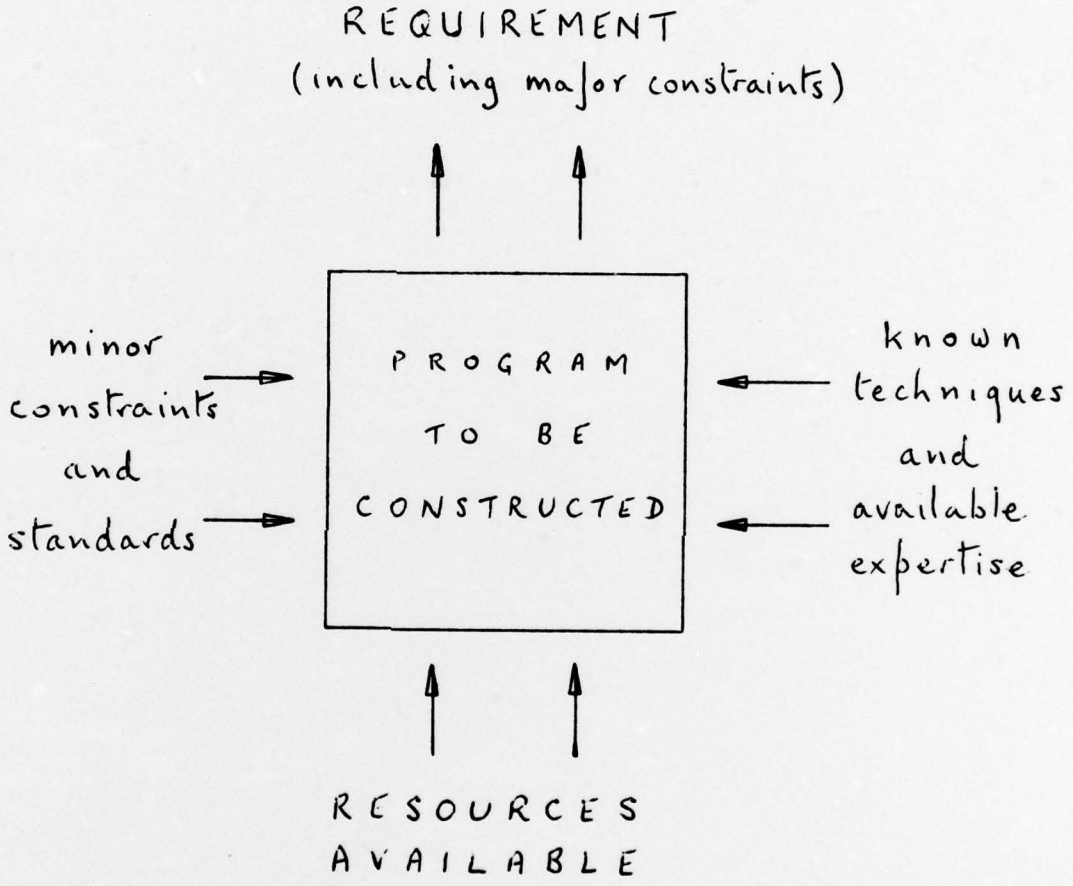


Fig.3 The programmer's problem

REPORT DOCUMENTATION PAGE

Overall security classification of this page

UNLIMITED

As far as possible this page should contain only unclassified information. If it is necessary to enter classified information, the box above must be marked to indicate the classification, e.g. Restricted, Confidential or Secret.

1. DRIC Reference (to be added by DRIC)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification/Marking
	RAE TM Math 7604	N/A	UNLIMITED

5. DRIC Code for Originator	6. Originator (Corporate Author) Name and Location
850100	Royal Aircraft Establishment, Farnborough, Hants, UK
5a. Sponsoring Agency's Code	6a. Sponsoring Agency (Contract Authority) Name and Location
N/A	N/A

7. Title

An essay on computing

7a. (For Translations) Title in Foreign Language

7b. (For Conference Papers) Title, Place and Date of Conference

8. Author 1. Surname, Initials	9a. Author 2	9b. Authors 3, 4	10. Date	Pages	Refs.
Gilbey D.M.			July 1976	23	-

11. Contract Number	12. Period	13. Project	14. Other Reference Nos.
N/A	N/A		

15. Distribution statement

(a) Controlled by –

(b) Special limitations (if any) –

16. Descriptors (Keywords) (Descriptors marked * are selected from TEST)

17. Abstract

An essay is presented on the nature and difficulties of working on or with computing systems. It is hoped that the intelligent layman can read past any unfamiliar jargon and still be convinced, along with the 'occasional programmer', that modern computing is mainly about the creation and management of the complex.

F5910/1